Computer Systems and Networks

ECPE 170 – Dr. Pallipuram– University of the Pacific

Build Tools + Makefiles

These slides are credited to Dr. Shafer

Lab Schedule

Activities

- **This Week**
  - Intro to Build Tools and Makefiles
  - Intro to C 1
  - **Lab 3 – Build Tools**
- **Next Week**
  - Intro to C 2
  - **Lab 4 – C Programming Project**

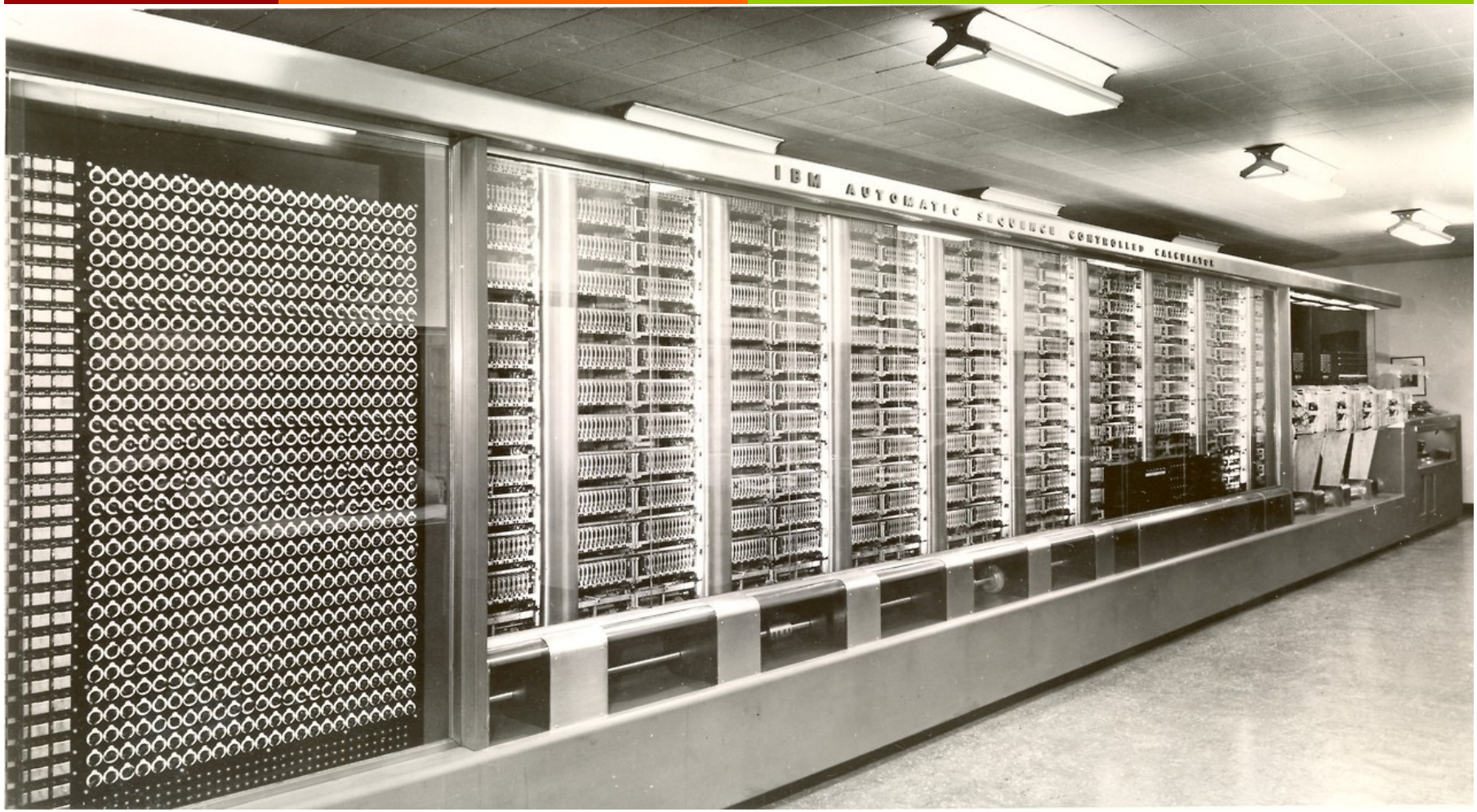
Deadlines

- **Lab 3 – Sep 13th 2021 by 11:59 PM**
- **Lab 4 – Sep 28th 2021 by 11:59 PM**

Person of the Day: Dr. Grace M. Hopper

- Ph.D, Mathematics, Yale, 1934
- Enlisted US Navy during WWII
- Early programmer of Harvard Mark I computer
- Developer of *first* compiler
- COBOL designer
- Caught the first computer "bug"
- Rear Admiral, USN

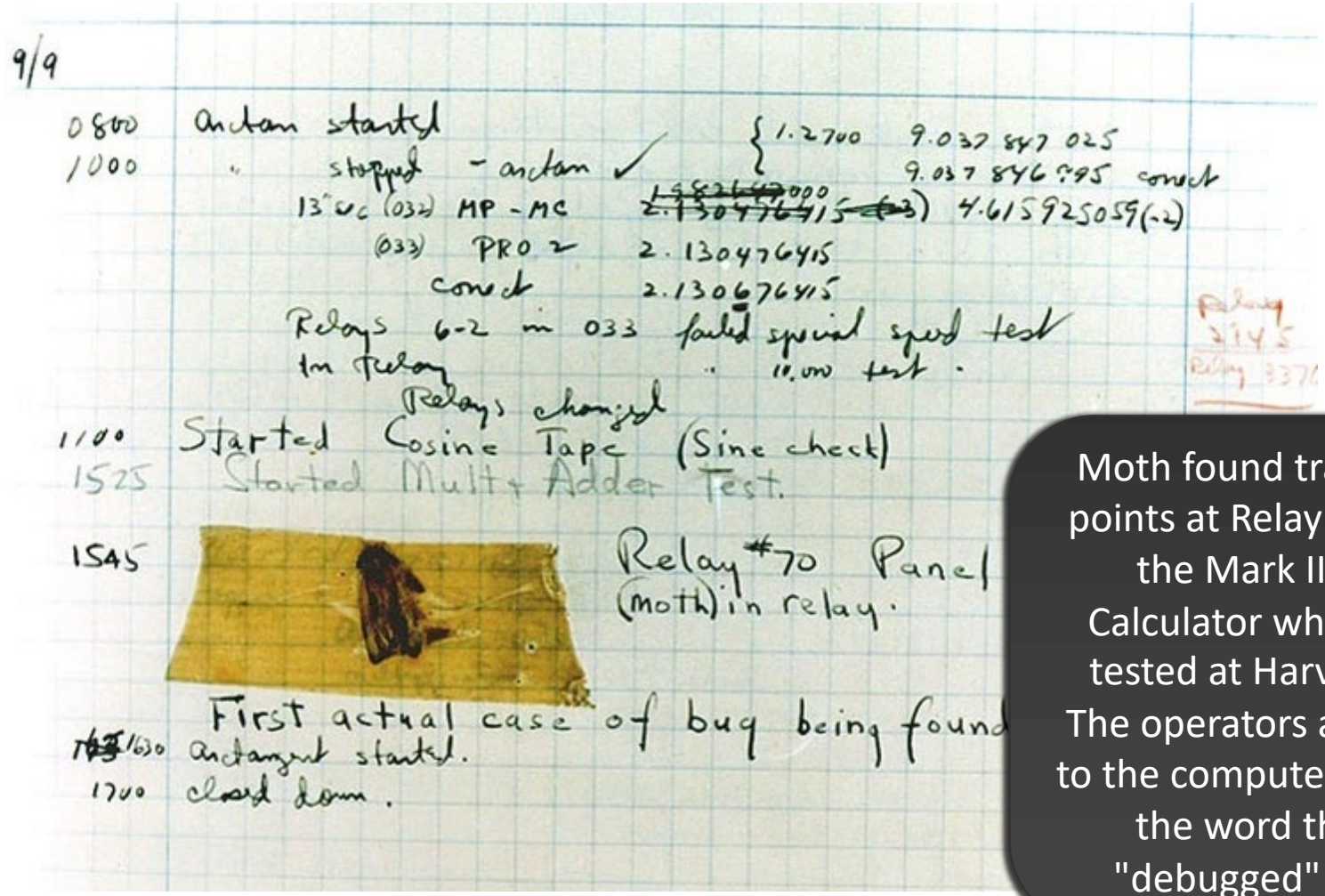
Grace Hopper - *Harvard Mark I (1944)*



Grace Hopper - *Harvard Mark I (1944)*

- Electro-mechanical computer
- 51 feet long, 8 feet high
- 10,000 pounds
- Punch tape program input

Grace Hopper – *Computer Bug (1947)*

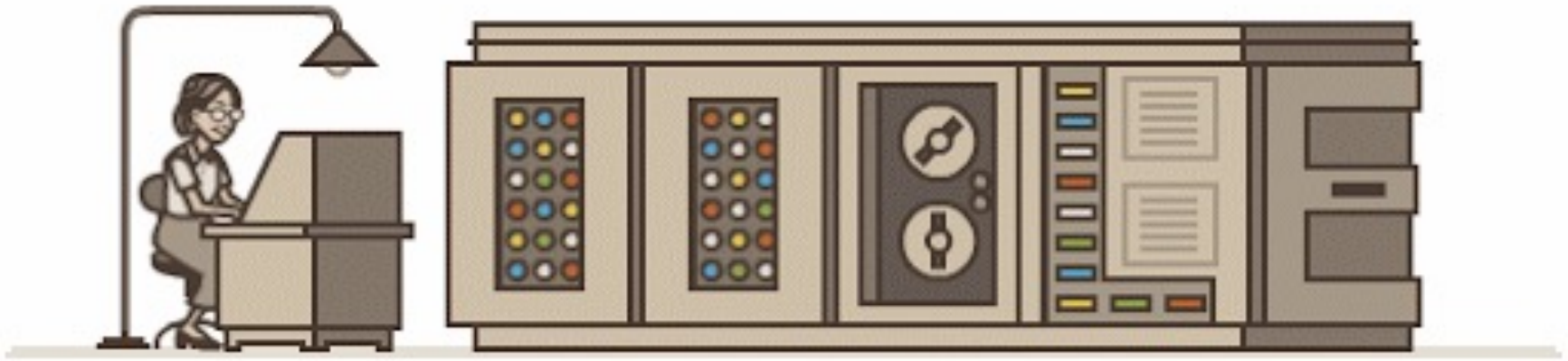


Moth found trapped between points at Relay # 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University. The operators affixed the moth to the computer log and put out the word that they had "debugged" the machine.

Grace Hopper - *A-0 Compiler (1951)*

- First compiler ever written for electronic computer
  - UNIVAC 1 computer
- Write your code as a collection of subroutines
- A-0 compiler combined the subroutines into one machine code program
  - More like a “linker” today…

Grace Hopper – *COBOL Language (1959)*



Tribute:
Google Doodle

Tribute: Cray XE6 "Hopper" at NERSC

Person of the Day: Richard Stallman

- Founder of
  - GNU project – “GNU’s not Unix”
  - Free Software Foundation
- Author
  - GNU C Compiler (GCC)
  - Emacs text editor
- GNU Manifesto
  1. Freedom to run a program for any purpose
  2. Freedom to study the mechanics of the program and modify it
  3. Freedom to redistribute copies
  4. Freedom to improve and change modified versions for public use

Person of the Day: Richard Stallman

- “Steve Jobs, the pioneer of the computer as a jail made cool, designed to sever fools from their freedom, has died.

  As Chicago Mayor Harold Washington said of the corrupt former Mayor Daley, "I'm not glad he's dead, but I'm glad he's gone." Nobody deserves to have to die — not Jobs, not Mr. Bill, not even people guilty of bigger evils than theirs. But we all deserve the end of Jobs' malign influence on people's computing.

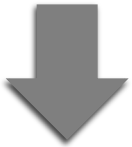
  Unfortunately, that influence continues despite his absence. We can only hope his successors, as they attempt to carry on his legacy, will be less effective.”

  - Richard Stallman, 10/6/2011

Toolchain

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
```



```
unix>  ./program
hello, world
```

Behind the Scenes

- **Motivating Question**
  - **What really happens between typing in the “Hello Word” program, and seeing the output on the console?**

Pre-Processor

- **Think of this as a "find and replace" wizard for your source code**
- **Include header files**
  - Literally insert .h file lines into .c file
- **Macro expansion**
  - Macro = fragment of C code
    - `#define IS_POSITIVE( _x ) ( _x > 0 )`
  - Preprocessor replaces macro with original definition in source code
- **Conditional compilation**
  - Include or exclude parts of the program
  - `#ifdef CONTROL`


THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."
HEY! GET BACK TO WORK!
COMPILING!
OH. CARRY ON.

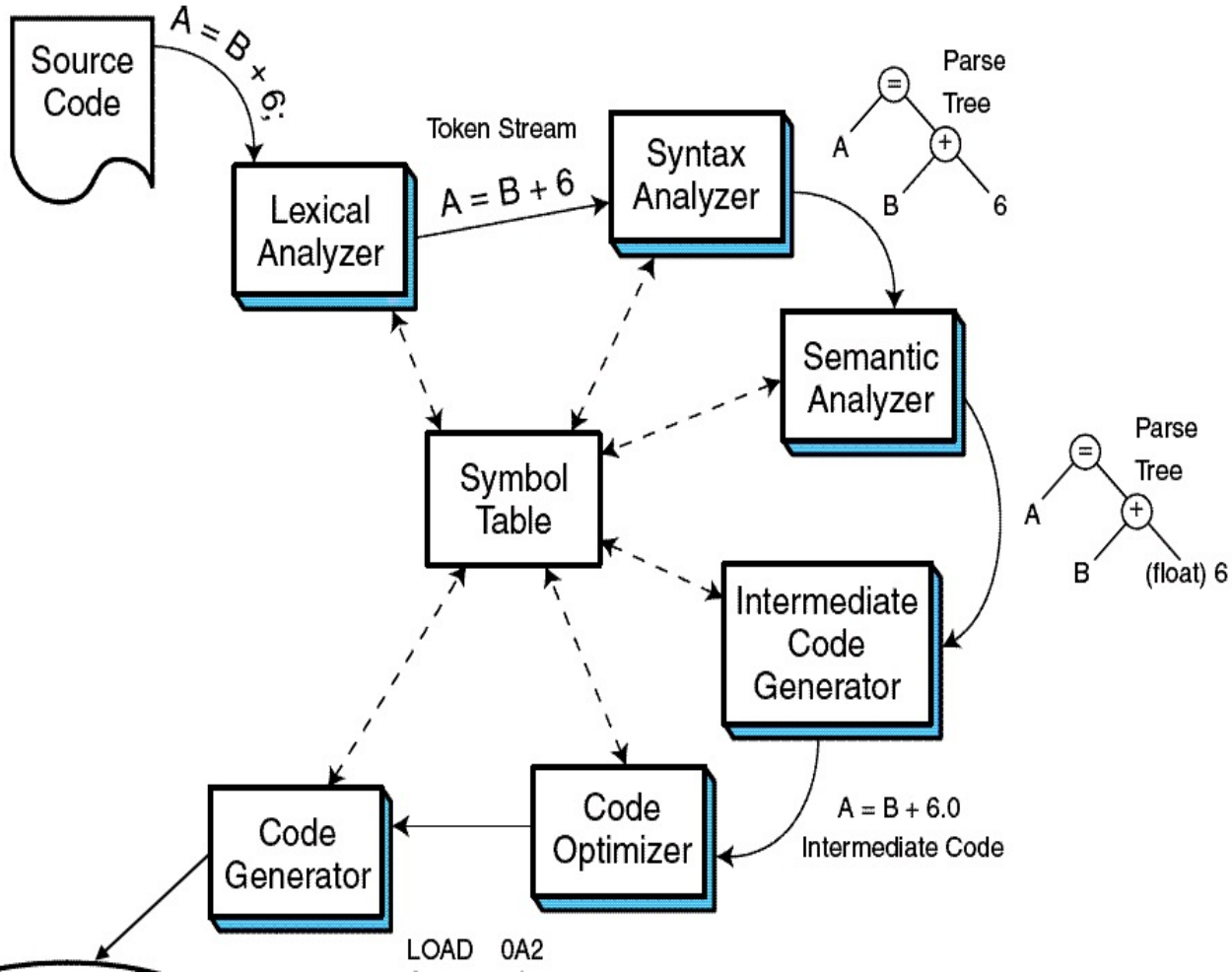
Compiler

- Basic goal
  - Input: **High-level language source code**
  - Output: **Machine code** for processor family
- 6 steps to accomplish transformation
- Steps 1-3 – source code analysis:
  1. **Lexical analysis** extracts tokens, e.g., reserved words and variables
  2. **Syntax analysis** (parsing) checks statement construction
  3. **Semantic analysis** checks data types and the validity of operators

Compiler Operation

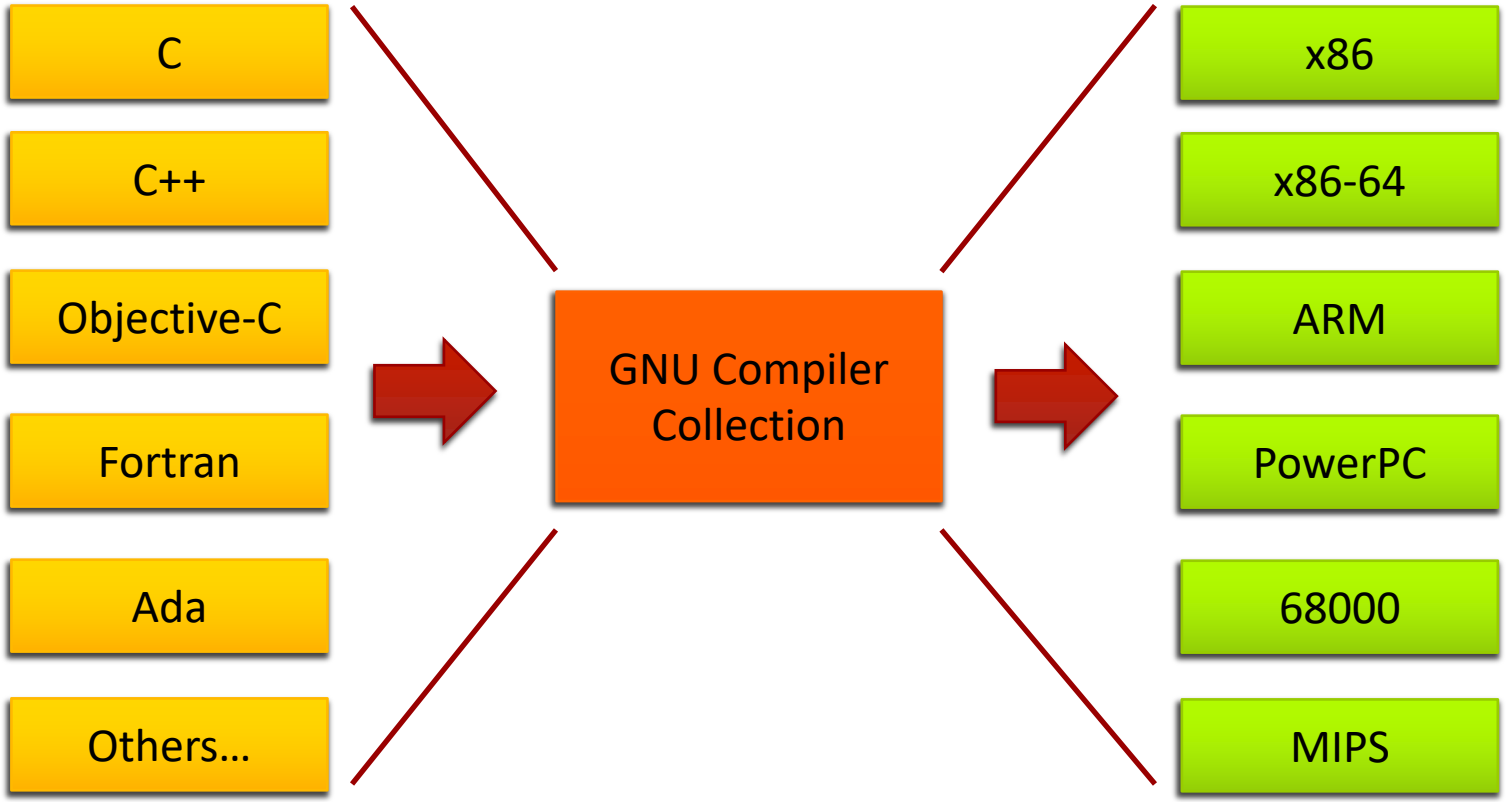
- Steps 4-6 – **Synthesis** phases:
  4. **Intermediate code generation** creates three address code ("fake assembly code") to facilitate optimization and translation
  5. **Optimization** creates (real) assembly code while taking into account architectural features that can make the code efficient
  6. **Code generation** creates binary code from the optimized assembly code
- We write these steps as separate modules
  - Benefit: Compilers can be written for various CPU architectures by rewriting only the last two modules

Compiler Operation



Why So Many Compilation Steps?

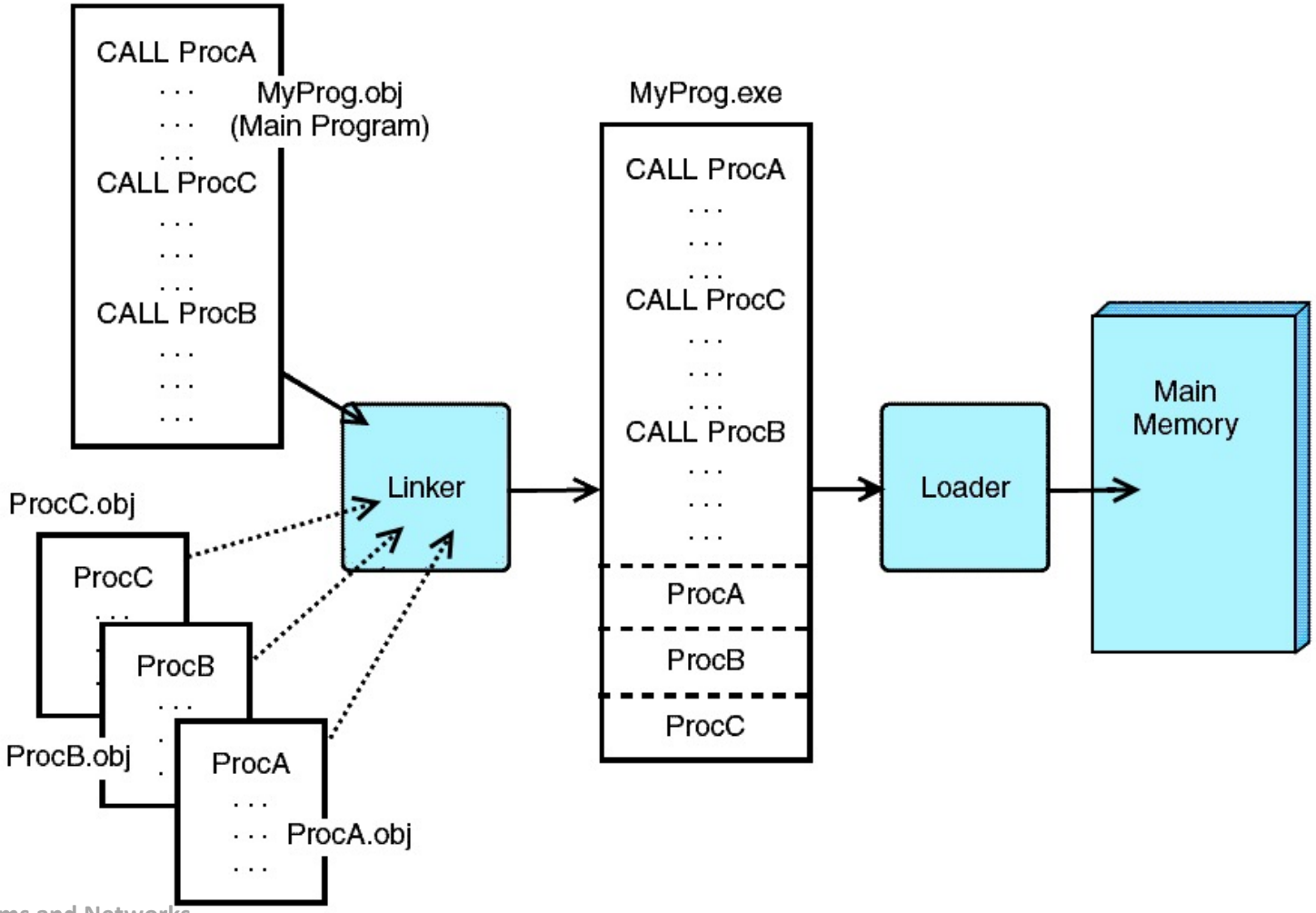
We don't *just* care about 1 language or 1 processor family!



Linker

- Real programs are typically written with **multiple source files** and many subroutines
  - Each file is compiled separately
  - But we need some way to join everything together into a single executable file
- This is the job of the **linker** (aka “link editor”)
  - Input – many files with binary machine code
  - Output – single file with all of the necessary binary machine code

Linker + Loader



Result: Program binary (saved on disk)

```
1101111010100000010100000011011110101000
0001010000001101111010100000010100000001
1011110101000000101000000110111010100000
0010100000011011110101000000101000000011
0111010100000010100000011011110101000000
0101000000110111010100000010100000011000
1110101000000101000000110111010100000000
1010000001101111010100000010100000001101
1101010000001010000001101111010100000001
```

Shell / GUI

- User instructs computer to run program
  - Shell command?
  - Mouse / keyboard action in GUI?

Operating System

- Security: OK to run file?
- Memory management: Find space and create new virtual memory region for this program
- Filesystem: Retrieve program binary code from disk
- Loader: Place program binary code into memory
- Scheduler: Find CPU time for program to run
- Context switch – Program starts running

Makefiles – Lab 3

Makefile

- Goal: Compile our program with one command:

```
unix>  make
```

- Challenge
  - Every program is different!
  - Different source files, different compilers / settings, different external libraries, etc…
- A **Makefile** is a **text file** that specifies how to compile your program
  - The `make` utility reads the Makefile
  - You'll learn how this file works in Lab 3